Technical Report
CMU/SEI-93-TR-13
ESC-TR-93-190

2

Software Product Liability

Jody Armour
School of Law, University of Pittsburgh

Watts S. Humphrey
SEI Fellow, Software Engineering Institute

August 1993

Technical Report
CMU/SEI-93-TR-13
ESC-TR-93-190
August 1993

Software Product Liability

Jody Armour

School of Law, University of Pittsburgh

Watts S. Humphrey

SEI Fellow, Software Engineering Institute

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/ENS
Hanscom AFB, MA 01731-2116

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.

This report was funded by the U.S. Department of Defense.

Table of Contents

Software Product Liability

Abstract: Voyne Ray Cox settled into the radiation machine for the eighth routine treatment of his largely cured cancer. The operator went to the control room and pushed some buttons. Soon, the machine went into action and the treatment began. A soft whir and then an intense searing pain made him yell for help and jump from the machine. The doctors assured him there was nothing to worry about. What they didn't know was that the operator had inadvertently pushed an unusual sequence of controls that activated a defective part of the software controlling the machine. He didn't die for six months but he had received a lethal dose of radiation. This software defect actually killed two patients and severely injured several others. The final decisions in the resulting lawsuits have not been made public.

Software defects are rarely lethal and the number of injuries and deaths is now very small. Software, however, is now the principle controlling element in many industrial and consumer products. It is so pervasive that it is found in just about every product that is labeled "electronic." Most companies are in the software business whether they know it or not. The question is whether their products could potentially cause damage and what their exposures would be if they did.

While most executives are now concerned about product liability, software introduces a new dimension. Software, particularly poor quality software, can cause products to do strange and even terrifying things. Software bugs are erroneous instructions and, when computers encounter them, they do precisely what the defects instruct. An error could cause a 0 to be read as a 1, an up control to be shut down, or, as with the radiation machine, a shield to be removed instead of inserted. A software error could mean life or death.

1 The Software Liability Problem

Software product liability has not been a historical problem for four reasons. First, until recently software has largely been used by experts in the computer departments of large corporations. Only in the last few years have small businesses and the general public used it directly. Second, the design of many early software-controlled products has remained relatively simple. The nuclear power industry, for example, has opted for a very conservative design philosophy (see Box A). Third, the leading vendors have historically marketed their products under tight contracts. Finally, until recently, there have been few lawyers with the expertise to handle such cases.

As the general use of software and software controlled products grows and as the public exposure to poor quality products mounts, the product liability problem will increase. There is, in fact, evidence that this is happening already. Marr Haack of St. Paul, the insurance underwriter, reported that of several hundred software-related claims, half concerned software development and quality. John Moore, of insurance underwriter Shand Morahan, reports that their suits principally come from failure to perform or failure of the installed software to function.

Tom Cornwell of Chubb estimates that in three years, their claim size for software-related business has doubled. John Lautsch, a Silicon Valley attorney, further reported that the ABA's computer law division membership grew from 89 members in 1980 to more than 500 members in 1985. Membership has continued to increase and it now has passed 1100. There is, in fact, a growing body of legal experience with software damages.

The fact that few of these claims are for personal injury or physical damage might suggest that software is a low risk product. If it continued to be used as in the past, that could well be true. There is, however, growing evidence that this will not be the case.

2 The Increasing Uses of Software

We all expect computer-related products like operating systems, telephone switching systems, and banking systems to contain millions of computer instructions. Less well known is the fact that common products such as TV sets, radios, VCR's, and telephones contain hundreds and even thousands of program instructions. Projections are that such devices will soon contain a million or more program instructions. The current rate of code growth in many ordinary consumer products, if extended for ten years, will approach 100 times. Software use is exploding and it is exploding fastest in products that are used by the general public.

From an industrial perspective, software is an almost ideal product. While many decry software's high cost and poor quality, it is the lowest cost and highest quality way to provide sophisticated product function. In fact, often software is the only feasible way to implement these functions. Once software is developed, it doesn't wear out or deteriorate, it can be legally protected, its manufacturing costs are trivial, and it is economical to modify and enhance. A good example of these benefits is the nuclear power industry. Many of the nation's 113 nuclear power plants still have controls that are 30 or more years old. As the industry replaces them with modern software-controlled devices, they find reliability is much improved.

The exposure to product liability is greatest when software is used to control sophisticated operations in safety-critical situations. Witness, for example, the following list of software errors that resulted in recalls of medical equipment:

- Incorrect match of patient and data
- Incorrect readings in an auto analyzer
- Faulty programming provided false cardiac FVC values
- Faulty programming caused pacer telemetry errors
- Incorrect software design caused lockup of cardiac monitor
- Incorrect calculations
- Failure of central alarm in arrhythmia monitor
- Table top moved without a command
- Detector head could hit the patient
- Algorithm error caused low blood pressure readings
- Over infusion due to programming error

These are only a few examples from a much longer list.

While software is arguably the highest quality product made by mankind, without great care, it can easily become among the most complex. The competitive demand for more and more product functions leads to more and more program instructions. As program size grows, sc does its complexity and it is this combination of size and complexity that leads to problems with software quality. Given enough instructions, even software products that are considered high quality by today's standards will contain many defects. While many such defects are likely simple bugs, some could be major functional errors and omissions. Since any defect can cause user problems, it is clear that the software community must improve software quality faster than it expands product size. The issue is not whether software is safe but whether it is used in safety critical systems. If it is, with the current state of software practice, any software is potentially unsafe.

3 Software Liability Strategies

Suppliers can protect themselves from legal liability for defective software by investing in improved product quality or by relying on legal protections. These strategies are not mutually exclusive. By the time someone is injured by poorly performing software, it is too late to fix the real problem. If the software your organization produced caused the injury, you will likely pay, and perhaps a lot. It is far safer and less expensive to avoid the problem than to attempt to limit damages after the fact. While problem prevention takes work and is not free, in the long run it costs a lot less than the costs of fixing the problems. With software, it turns out, problem prevention actually saves money. Data shows that software defect repair is as much as 100 times more expensive than defect prevention.

One reason why reliance on legal protection is problematical is the high cost of winning. While there is little data on software to date, Beech, the aircraft company defended and won most of their 203 personal injury actions over a four year period. They spent, however, an average of $530,000 to fight each case. An insurance company reports a recent software liability case that cost $3,000,000, and that was to win. It would have cost much more had they lost. While no one expects the software industry to face large volumes of litigation soon, current trends are increasing.

Improving the quality of software development and maintenance is the best long-term strategy. Since this can take time, however, software process improvement should be coupled with a product liability strategy. Figure 1 shows the logic of software product *liability litigation*. Here, the three types of recovery are strict liability, negligence, or product warranty. A complainant will generally prefer to recover in this order while vendors will prefer the reverse. While an injured party can pursue any one or more of these strategies, the supplier must have a defense against them all. It should even be assumed that clever complainants will attack the weakest point in the supplier's defenses. Often, in fact, litigation strategies are devised by knowledgeable ex-employees who have become involved in the case. The problem is to understand how to influence these strategic outcomes by current actions.

4 Strict Liability

Strict liability is that part of tort law that covers damage caused by or threatened by unreasonably dangerous products. In contrast to negligence, which focuses on the processes used to produce products, strict liability focuses on the product itself and whether or not it contained one or more unreasonably dangerous defects. The first question then is the likelihood that your product will be judged unreasonably dangerous. While this may not seem a serious concern to most companies, somewhat circuitously, courts deem any defective product which threatens physical harm to person or property to be "unreasonably dangerous." So the key issue is whether software is a product.

While software is essentially information, courts may consider information to be a product. In one case, for example, a court applied the strict liability doctrine to aircraft instrument approach charts that contained fallacious data. In 1991, the 9th U.S. Circuit Court of Appeals ruled a publisher not liable for material in a book on mushrooms. In discussing criteria for considering information a product, however, they referenced the aeronautical charts case as one example and added that "Computer software that fails to yield the result for which it was designed may be another." Thus, there is good reason to believe that, for liability purposes, software could well be treated like a product.

Customer-supplier transactions frequently involve both sales of products and the rendering of services. Offerings to develop, install, service, or operate software would naturally be viewed as services, but the software itself would most probably be considered a product. While the courts have developed tests to deal with these sales/service hybrid transactions, to date they have generally considered software a product and applied the doctrine of strict liability. Moreover, if a defect merely threatens harm to person or property, the supplier may be strictly liable for purely economic losses even though there was no actual physical injury. To be safe, organizations should thus treat software as a product.

Thus, if a software defect threatens the person or property of a customer or a third party, the injured party is entitled to bring a strict liability claim against the supplier notwithstanding contractual disclaimers, limitations of remedies, and limited warranties. The virtue of a strict liability action from the claimant's perspective is that there is no need to prove that the supplier was negligent. If the product was defective and it caused the claimant's loss, the claimant wins without further ado.

There is some debate about whether software is a product or a service. There is growing evidence, however, that courts will consider software a product. Further, if your products are transportation, medical, construction, or farming equipment, an improperly executed product function could have lethal or at least physically harmful consequences. While this is a serious exposure for those involved, it is not the general case for most organizations with software in their products. Thus, the requirement of actual or threatened physical damage means that this outcome is a relatively low corporate exposure except for those who supply potentially dangerous products.

5 Negligence

Negligence is defined as conduct that falls below the standard established by law to protect persons against unreasonable risk of harm. Under negligence, a supplier is not responsible for every software defect that causes customer or third party loss. Responsibility is limited to those harmful defects that it could have detected and corrected through "reasonable" quality control practices. It is the supplier's failure to practice reasonable quality control that constitutes negligence and that causes the liability exposure.

While the proof obligations for negligence can be difficult, there is no need for actual or imminent physical damage. Economic or even intangible losses are sufficient. Since contracts can be written to protect against supplier negligence, one might think that this situation is only of concern where there is no contract. It turns out, however, that contracts between corporations and individuals can often be seen as between unequal parties and thus judged unconscionable. Negligence is thus the area of greatest risk for organizations with software-intensive products.

Negligence awards for physical damages can be large but those for economic losses like reduced profits or missed business opportunities can be enormous. These losses could even be completely disconnected from any personal injury or property damage claims.

Thus, it is likely that suppliers will be liable for customer damage caused by negligently developed and maintained software. In fact, *the courts do not even require* that the complainant be a customer. Third parties whose businesses have been disrupted by a defendant's negligence may even recover for their lost profits and missed business opportunities. There is little case law as yet on the recovery of purely economic losses due to defective software. Judged in the broader context of negligence case law, however, such recoveries seem likely. Further, the complainant's recourse against a negligent supplier applies whether the offering in question is a service or a product.

Especially where physical damage is involved, courts may disregard a contract in which a customer expressly assumes the risk of the supplier's negligence. Negligence, when proven, can thus surmount almost any legal obstacle the suppliers erect. As courts and legislatures become less tolerant of negligent corporate behavior, any business that is judged negligent will likely pay for actual damages, possibly be assessed punitive damages, and may even face regulatory or criminal proceedings. Therefore, the only secure protection is thus not to be negligent.

6 Warranties

Warranties contractually assure customers that the products they buy will perform as stated. When properly constructed, they also limit the supplier's liability in the event of nonperformance. Historically, for example, software suppliers have contracted to deliver a package of material that contains some software. This software is warranted to run on a given machine configuration but no assurance is given as to what that software will do.

Explicit software contracts can offer very flexible warranty protection but also have limitations. To win a warranty claim, the plaintiff must have a valid contract that the supplier did not fulfill. Demonstrating this can be a daunting challenge for a complainant, particularly when the suppliers write the contracts. Further, even when the case can be proven, the damages are generally limited to the monies paid for the product or to some amount of liquidated damages specified in the contract. Short of total vindication, this is the preferred corporate case. Unfortunately, in dealings with the public, it is not an outcome that can be guaranteed.

In determining liability exposure, the first question to examine concerns your contract practices. Do you regularly use explicit contracts with your customers? As your lawyers will explain, most companies are operating under contracts even when they don't think they are. Salespersons' comments, invoices, shipping labels, and even advertising can be part of a contract unless you make sure they are not. Since proven misrepresentation can void just about any contract, plugging this defensive gap with limited warranties and disclaimers should be a top priority.

Contract law involves the Uniform Commercial Code (UCC). This is an agreement between all the states (except Louisiana and Washington, D.C.) that specifies how to interpret contracts. The UCC treats, among other things, the implied warranties of fitness for a specific purpose and merchantability. The warranty of fitness for a specific purpose arises whenever the buyer relies on your expertise in selecting a product to perform a particular function that has been described to you. The key point is that here your product has an implied contractual warranty to do what your customer said was wanted. Thus, a warranty of fitness for a particular purpose is an implied promise by you that your software will meet the needs your customer communicated to you.

In contrast, warranties of merchantability accompany the sale of goods irrespective of any communication between buyer and seller. These provisions are imposed by the law as the basis for interpreting your overall performance under the contract. Basically, to be merchantable, your software must be of the general kind described and it must be reasonably fit for the general purpose for which it was sold.

Even if your contracts specifically exclude the commitments of merchantability and fitness for a specific purpose (as they can), you could still have a problem. If the courts judge that you were an expert and your customer was not, they could void the contract as between unequal parties and thus unconscionable. You had special knowledge and were thus obligated to protect your customer's interests.

While there is a lot more to contract law than this, the basic issue with software is that the supplier is generally an expert on an arcane and sophisticated technology and the customer is not. Unless you take extraordinary legal and marketing precautions, you contracts may not protect you.

So far we have talked about the good news. The bad news is that you may not even have a contract with the injured party. Even if all your products are covered by iron-clad contracts, the injured person could be a third party and not be covered. If, for example, you sell through distributors, your contracts are with the distributors and they deal with the users. If your contracts hold the distributor responsible for customer claims, you will likely have few distributors. Further, even then the users could reach you through the courts. You would then have the unappetizing task of suing your distributors. In any event, in dealings with third parties, you have no contractual protection and you are at the mercy of the court's interpretation of tort law.

7 The Current State of Software Practice

Software is a relatively new technology. The early software concepts are over a hundred years old but the first significant computer programs were only written in the early 1950s. Since then there have been many advances. The processes for building software, however have not made much progress. The methods most software developers use today are thus much like those used 30 and 40 years ago. The programmers start with a general understanding of the problem and then solve it in an individualistic and often highly creative way. The resulting products must then be tested to find and correct their many latent defects.

This build, test, and fix quality technology is almost universal for software. No other modern technology considers this even a minimally acceptable approach. For example, no self-respecting semiconductor engineer would consider testing and fixing all the defective chips coming off the production line. Just as Drs. Juran and Deming taught and the Japanese have amply demonstrated, quality must be built into products from the beginning. Testing is no substitute for proper design in either hardware or software.

The current state of software quality control can be best understood by examining the current state of software practice. The Software Engineering Institute of Carnegie Mellon University has developed a way to assess the capability of software organizations (see box B). On a scale of 1 (worst) to 5 (best), over 80% of the software organizations studied were at level 1. Less than 20% were at level 2 and very few were at levels 3, 4, and 5 (see Box C). This means that most software organizations have poor project management practices, miss schedules and costs, and deliver products of unknown, but often poor, quality. As this field develops, it is thus likely that courts will deem a supplier's failure to move beyond SEI maturity level 1 to be negligence.

Very few groups have achieved SEI level 5. For example, the software that flies the Space Shuttle was developed by a project rated 5 by NASA. This is an IBM group in Houston Texas that has spent many years improving their development methods. To date, in 19 years supporting the shuttle, their software has not had a single mission-critical defect. As this demonstrates, the quality benefits of a mature software process are substantial.

8 Some Example Cases

To illustrate software product liability principles, consider three hypothetical examples:

8.1 Case 1: A Tax Program

You manufacture and sell, through distributors, a general purpose program to assist small companies in handling their taxes. You provide the program with a limited warranty that offers to refund the users' payments if the product is defective.

One user claimed that data provided by your program misled him into making an unsound investment that cost him a substantial amount of money. On investigation, you found that there was a bug in the program that could have produced the fallacious information your user claims. What are the legal consequences?

You would obviously try to get the user to accept a refund of the $450 he paid for the program in return for a full release from any further liability. While worth a try, this strategy is not a guaranteed success.

Your user, who did not buy the program directly from you, could not claim under a contract. Also, since no physical injury or property damage was involved, claims cannot be made under strict liability. The final recourse, therefore is to claim negligence. Here, the question is: did you follow best industrial software development practices to assure that such problems did not occur? If the court can be convinced that you did, you would likely win. If not, you could pay substantial damages.

8.2 Case 2: A Computerized Drafting System

You manufacture, sell, and service an advanced computerized system for producing architectural drawings and specifications. The system is program controlled with a range of optional and custom features. You sell it under a warranty that limits your liability to five times the total moneys paid for the system.

One of your customers claims that he bought your system expressly to complete a rush project and that program defects severely delayed his work. He missed his committed dates, forfeited a substantial incentive payment, and lost money on the architecture contract. He claims defects in your software caused several files to be garbled, necessitating extensive rework. On investigation, you find that a software defect could have caused the alleged problem. What are the legal consequences?

Since there have been no personal injuries or property damage, strict liability is not involved and the issues concern negligence and warranty. While you are not anxious to pay the warranty maximum of five times the $9,500 paid for the system, you want to avoid having to pay the total claimed damages of $450,000. Your strategy is thus:

1. You first claim that he was responsible for the failure to deliver on schedule, not your system.
2. You will next claim that the warranty limits your liability to $47,500.

Your customer will first assert that you negligently designed, developed, and supplied the system software and that the contractual limitations are thus not valid. Your defense is to show that you exercised reasonable care in developing and testing the software.

If his negligence claim fails, your customer will next claim that you, the expert, misled him, the neophyte, about the system's capabilities. Thus, the contract is thus not valid and the sales claims were guarantees. Here, you argue that your customer is knowledgeable and that you made no invalid claims about the system's capabilities.

Finally, the customer could claim under the contract that your system did not perform as promised. If your product actually had the defect claimed, you could well pay the contractually limited damages.

8.3 Case 3: An Automated Tunneling Machine

You manufacture and market a sophisticated computerized drilling machine that senses underground conditions while drilling tunnels. It is designed to determine the structural strength of the strata through which it drills and to keep the miners informed about mine conditions. Your machine is marketed under a warranty that limits your liability to ten times the moneys paid for the machine.

One of your machines was used in a coal mine to dig a tunnel where there was a fire, a collapse, and a loss of life. While no one survived to tell what happened, the miners' families claim that your machine was defective because it did not alert them to the presence of methane or the likelihood of a collapse.

Here, the argument concerns strict liability. If your machine contributed to the damages, you will almost certainly be held liable. If it did not, you will probably have no liability. The issue could thus revolve around the likely behavior of your machine and whether it could have caused the alleged accident either through its design or through a malfunction. Experts would likely be used to examine your product to see if there were any plausible ways that it could have contributed to the accident. Here, sound design methods, state-of-the-art quality practices, and comprehensive testing are your best defense.

In every one of these example cases, poor software quality practices can be a serious disadvantage. This is particularly true if the practices of your software people do not at least equal the best in your industry. If they do not, your exposures could easily exceed your contractually limited liability.

9 The Improvement Opportunity

If you are in the business of supplying potentially hazardous products, you should take immediate short-term preventive action. First, make sure your people use the best available methods to assure that your products are safe. Nancy Levison, at the University of California, devised methods for determining whether complex software could be unsafe. Rather than look for defects or bugs, she suggests that developers rigorously search for all potential ways the total system could be dangerous and then design special hardware or software provisions to insure these conditions never occur. That way, even if there is a software defect, other blocks will insure there is no damage. This extremely effective technique should be used by organizations concerned about the safety of their hardware-software systems (see Box D).

Longer term, you must address product quality. As noted above, this requires an intensive and continuous focus on software process improvement. The limited data now available indicates that there is as much as a 1000 to 1 improvement in quality between organizations at SEI level 1 and SEI level 5. While it takes many years to advance to level 5, many groups have improved to levels 2 and 3. Hughes Aircraft, for example, reaped a 5 to 1 return in just one year on their process improvement program to reach level 3. They also cut their late software deliveries to almost zero. Motorola, Israel, reports that their first two products to reach SEI level 2 met their cost and schedule targets and have had zero customer-reported defects in their first 6 and 10 months of use. Raytheon has similarly reduced defects while saving over 7 times their quality improvement costs. These companies and many others have established and funded substantial continuing programs to improve their software processes (see Box E).

The challenge is thus to improve the maturity of your software processes so they consistently produce high quality products. The suggested steps are:

1. Act rapidly to determine the maturity of your software process.
2. If it is low, as it probably is, take immediate and aggressive improvement action (see Box E):
   - launch and maintain a permanent emphasis on software process improvement,
   - utilize the best state-of-the-art development and test practices,
   - utilize applicable new technology developments,
   - maintain a quality distribution and support system.

3. Under the guidance of experts, institute design-for-safety practices.
4. Improve your contracts:
    - get competent legal advice,
    - be sure your customers are aware of **all** product risks,
    - use clear and reasonable warranties,
    - meet your commitments.
5. Until you have taken these steps, avoid delivering complex software to high risk markets (see box A).

10 Conclusion

Ultimately, any successful strategy must reduce the risk of public injury by defective software. This requires that you establish and maintain a mature software process. This can take time. Starting at SEI level 1, it could take three to five years just to reach level 2. Each succeeding level can take another two to three years. For a level 1 organization, this is a sustained 9 to 14 year improvement effort. Substantial benefits, however, accrue with each improvement step. The best comparable example of a long-term corporate improvement strategy is the Japanese drive for quality following World War II. While it took them over 20 years, they netted leadership in watches from the Swiss, cameras from the Germans, and automobiles, steel, and shipbuilding from the U.S.

Quality is a long term issue and fortunately this is not an imminent crisis. By the time it is, however, it likely will be too late to limit the damage. Behind this risk, however, is an important long-term opportunity. An aggressive software process improvement program has helped many organizations to both improve their products and to save time and money. It could also help you to reduce your liability risks, improve product quality, and save money.

References

[Cohen 92]	Ilan Cohen, "Using SEI Maturity Model to Improve the Software Process," *IEEE Computer Society*, Sixth Israel Conference on Computer Systems and Software Engineering, Israel, June 2-3, 1992.
[Edwards 86]	W. Edwards Deming, *Out of the Crisis*. Cambridge. MA: MIT Center for Advanced Engineering Study, 1986.
[Dion]	Ray Dion, his new paper on process improvement at Raytheon
[Fowler 89]	Priscilla Fowler and Stan Rifkin, *Software Engineering Process Group Guide* (CMU/SEI-90-TR-24). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1989.
[Humphrey 91]	Watts S. Humphrey, Terry R. Snyder, and Ronald R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, (July 1991): 11- 23.
[Humphrey 89a]	Watts S. Humphrey, *Managing the Software Process*, Reading, Ma: Addison-Wesley, 1989.
[Humphrey 89b]	Watts S. Humphrey, David H. Kitson, and Timothy Kasse, *The State of Software Engineering Practice: A Preliminary Report*, (CMU/SEI-89-TR-1, DTIC ADA206573). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1989.
[Kitson 92]	David H. Kitson and Steve Masters, *An Analysis of SEI Software Process Assessment Results: 1987-1991* (CMU/SEI-92-TR-24). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.
[Kolkhorst]	Barbara G. Kolkhorst and A. J. Macina, "Developing Error-Free Software," Computer Assurance Congress '88, joint meeting with the IEEE Washington Section on System Safety and the Tri-Service Software System Safety Working Group.
[Levison 83]	Nancy G. Levison and P. R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering, SE-9*, 5 (1983): 569-579.
[Samuelson 93]	Pamela Samuelson, "Liability for Defective Electronic Information," *Communications of the ACM*, 36, 1 (January 1993).

Appendix A Software Design Complexity (Box A)

Software designs can be very simple or they can be extraordinarily complex. The decision of where a product should fall in this spectrum is traditionally made by designers who are properly striving to produce the best technical result. Unfortunately, their decisions of what is "best" are often not made with all the relevant information. For example, advanced technical methods generally increase complexity. Increased complexity increases the likelihood of error. This in turn increases the risk of potentially damaging development problems or product defects. In simplistic terms, the complexity progression for software-based systems is roughly as follows:

1. **Small single-function routines.** These software products are typically small enough so one developer can completely understand them and build them. Such products can also often be exhaustively tested so there is high assurance that they will do exactly what was intended.
2. **Large dedicated programs.** While generally too large for a single individual to develop or even to understand in detail, these programs are still reasonably testable. The reason is that they operate in a dedicated environment where the full range of operational conditions is often predictable and where a reasonably comprehensive set of tests can be devised. While a large amount of testing might be required, reasonably high product quality can generally be assured.
3. **Multi-programmed systems.** As the number of computer-controlled functions in a system increases, it is often possible to use one high-capacity computer to handle them all. This is done through a technique known as multi-programming where the computer spends a small fraction of its time handling each need. This technology has been found so effective that it is the basis for all of today's large-scale computer operating systems. As cost-effective and useful as this technique is, it greatly increases software complexity and vastly complicates testing. The reason is that, because of the interleaving of many independent jobs, it is practically impossible to devise comprehensive tests for all likely conditions. Such programs are thus subject to unpredictable behavior under unusual conditions.
4. **Multi-processing systems.** An even further step in complexity is often taken to improve performance and reliability. In situations where a failure is unacceptable, it is possible to build parallel systems with dynamic switching in the event of failure. These systems can be very complex, both because of the error detecting and switching logic and because of the increased number of job and system configurations that need to be tested. Generally, it is not possible to exhaustively test multi-processing systems.

This progression involves increasing software complexity to make more efficient and reliable use of hardware. For this trade-off to be effective, however, the software must be of very high quality. Prudent managers should thus consider this trade-off in selecting their product strategies and only use sophisticated software designs where the risk of damaging defects can be controlled.

Appendix B Software Process Maturity (Box B)

The Software Engineering Institute was established at Carnegie Mellon University in December 1984 to address the need for improved software in U.S. Department of Defense Operations. As part of its work, SEI developed the Software Process Maturity Model for use both by the Department of Defense and by industrial software organizations.

Software process maturity deals with the capability of software organizations to consistently and predictably produce high quality products. Maturity also implies that software process capability must be grown, which requires strong management support and a consistent long-term focus. The Software Process Maturity Model provides a graduated improvement framework where each level progressively builds on pnor process improvements. Because of its progressive nature, this framework can be used to assess software organizations and to define their most important areas for improvement. It also permits organizations to determine their relative standing with respect to other groups.

The five-level improvement model for software is shown in Figure B1. At the initial level (level 1), organizations typically operate without formalized procedures, cost estimates, or project plans. Schedules are late, cost targets are missed, and quality is unpredictable.

Organizations at the repeatable level (level 2) have learned that traditional engineering management works for software just as for other technical fields. These organizations have established basic software management practices: management oversight, product assurance, and configuration control. Such organizations can reasonably meet routine schedules and costs but newer technologies are often a problem.

Defined level (level 3) organizations have process specialists who maintain a focus on process improvement, keep management informed, and facilitate the introduction of new methods and technologies. These organizations have learned how to manage change and how to draw on industry experience to address new challenges.

Managed level (level 4) organizations have comprehensive quality and productivity measurements, a process database, and analysis and consultative resources to support their projects. They use data to statistically manage their work and they routinely set and meet aggressive quality goals.

At the optimizing level (level 5) the organization is focused on defect prevention and continuous process improvement. These World-class groups regularly set and meet more challenging productivity and quality goals.

Appendix C The State of Software Practice (Box C)

A principle use of the SEI Software Process Maturity Model has been in assessing software organizations to help them improve. This work has yielded substantial information on the state of U.S. software practice. While this data on several hundred projects is not a statistically valid survey, it does provide a reasonable indication of the problems in this field. The summary assessment data in Figure C1 shows that 81% of the organizations were at maturity level 1. Very few of these projects used even basic project management methods. The few projects at level 2 were in leading companies working on U.S. Department of Defense (DoD) contracts. Demanding DoD specifications typically forced them to use disciplined management practices.

The assessment data show that the principle areas level 1 organizations need to address are:

- Inadequate estimating and planning practices
- Poor control over commitments
- Inadequate or ineffective quality assurance functions
- Inadequate configuration and change management

The areas level 2 organizations need to address are:

- Insufficient or inadequate training
- Lack of focus on process improvement
- Continuing quality assurance problems
- Lack of quality data
- Inadequate testing

While these needs have some technical aspects, they are not technical problems and they have all been solved many times before. These findings clearly show that the principal process improvement needs of most U.S. software organizations concern management and not high technology.

Appendix D Software Safety (Box D)

Software safety concerns the risk of encountering software-caused hazards while using a system. Since software consists of electronic signals inside computers, it cannot directly cause harm. When software is improperly specified, designed, implemented, or used, however, it can cause systems to do damaging things. A software safety analysis must therefore consider the software in the context of the system it controls. A software safety analysis involves the following basic steps:

1. Perform a system hazard analysis to determine potential safety risks (see below).
2. Determine the system design and usage changes that should be made to prevent these hazards.
3. Implement these changes.
4. Test to assure that the changes prevent the hazardous events from occurring.

Hazard analyses can be performed in various ways but a common approach is called software fault tree analysis. It is done as follows:

1. Make a preliminary hazard analysis. This lists all the potentially unsafe actions that the system could conceivably perform. Even if there was no obvious way that software could cause such an action, it should be considered.
2. Construct a fault tree for each such hazard. The fault tree represents the logical relationships of all the conditions that would have to exist to cause the hazard to occur.
3. After the fault trees have all been constructed, determine the design or operational changes needed to guarantee that each hazard could not possibly occur.

Since poorly performed software safety analyses could easily overlook critical hazards, care must be taken. The procedure appears simple, but these steps can be quite complex and should be performed with the help of experts.

Appendix E Software Process Assessment (Box E)

Many successful software process improvement efforts have started with an assessment. This identifies the organization's unique problems and recommends steps to address them. Action plans are then established and implemented. After two or more years, another assessment is done and the cycle is repeated. Because of their pivotal role in process improvement, it is essential to handle the assessments properly.

An assessment is a diagnostic tool to aid organizational improvement. Its objectives are to provide a clear understanding of the organization's software practices, to identify key improvement areas, and to initiate improvement action. The assessment must start with the senior manager's commitment to support the assessment and the ensuing improvements. Software process assessments typically have the following six phases:

1. **Commitment.** Senior management, with line management agreement, commits to do the assessment.
2. **Selection.** An organization is selected to assist in the assessment. This should be a trained group that is competent to do such work[1]. An agreement is typically signed that commits management to supporting the assessment and implementing its recommendations.
3. **Preparation.** During this two to four month period, the assessment team is selected and trained, the assessment participants are identified, and the assessment mechanics are settled.
4. **Assessment.** The on-site assessment is conducted. This is an intense five day period where many project managers and professionals are interviewed. Its product is a thoroughly researched and reviewed set of findings that is presented to management and all assessment participants.
5. **Report.** An assessment report is next prepared that includes the findings together with recommendations to address them. Experience has shown that a written assessment report facilitates continuing process improvement.
6. **Assessment Follow-On.** This covers action plan preparation and implementation and should continue until the next assessment.

1. The SEI has trained and licenced a number of commercial organizations to perform assessments.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-93-TR-13	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-93-190
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO 63756E; PROJECT NO. N/A; TASK NO N/A; WORK UNIT NO. N/A

11. TITLE (Include Security Classification)
Software Product Liability

12. PERSONAL AUTHOR(S)
Jody Armour and Watts S. Humphrey

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	August 1993	24

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)

software product liability	safety critical systems
software	software development
software-controlled products	software practice

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Voyne Ray Cox settled into the radiation machine for the eighth routine treatment of his largely cured cancer. The operator went to the control room and pushed some buttons. Soon, the machine went into action and the treatment began. A soft whir and then an intense searing pain made him yell for help and jump from the machine. The doctors assured him there was nothing to worry about. What they didn't know was that the operator had inadvertently pushed an unusual sequence of controls that activated a defective part of the software controlling the machine. He didn't die for six months but he had received a lethal dose of radiation. This software defect actually killed two patients and

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631 22c. OFFICE SYMBOL ESC/AVS (SEI)

ABSTRACT — continued from page one, block 19

severely injured several others. The final decisions in the resulting lawsuits have not been made public.

Software defects are rarely lethal and the number of injuries and deaths is now very small. Software, however, is now the principle controlling element in many industrial and consumer products. It is so pervasive that it is found in just about every product that is labeled "electronic." Most companies are in the software business whether they know it or not. The question is whether their products could potentially cause damage and what their exposures would be if they did.

While most executives are now concerned about product liability, software introduces a new dimension. Software, particularly poor quality software, can cause products to do strange and even terrifying things. Software bugs are erroneous instructions and, when computers encounter them, they do precisely what the defects instruct. An error could cause a 0 to be read as a 1, an up control to be shut down, or, as with the radiation machine, a shield to be removed instead of inserted. A software error could mean life or death.